Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

ОТЧЕТ
ПО ПРАКТИЧЕСКОЙ РАБОТЕ №11
дисциплины «Алгоритмизация»
Вариант___

Выполнил:
Репкин Александр Павлович
2 курс, группа ИВТ-б-о-22-1,
09.03.01 «Информатика и вычислительная техника»,
направленность (профиль)
«Программное обеспечение средств вычислительной
техники и автоматизированных систем», очная форма обучения

_____
(подпись)

Руководитель практики:
Воронкин Р.А., канд. техн. наук,
доцент кафедры инфокоммуникаций

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2023 г.

Порядок выполнения работы:

1. Создана программа на основе приведённых в лекции примеров динамического программирования, для нахождения требуемого числа в последовательности Фибоначчи. Приведены три функции: fibTD, fibBU, fibBUImproved, затрачивающие одинаковое время $O = (n^2)$.

```
// Время выполнения = O(n*n).
static Integer fibTD(int length) {  3 usages
    if (numbers.get(length) == -1) {
        if (length <= 1) numbers.set(length, length);
        else numbers.set(length, fibTD( length: length - 1) + fibTD( length: length - 2));
    }
    return numbers.get(length);
}
```

Рисунок 1. Полученный код функции fibTD.

```
// Время выполнения = O(n*n). Итеративное решение с использованием массива.
static Integer fibBU(int length) {  1 usage
    ArrayList<Integer> values = new ArrayList<>();
    values.add(0);
    values.add(1);
    for (int i = 2; i < needed; i++) values.add(values.get(i - 1) + values.get(i - 2));
    return values.get(length);
}
```

Рисунок 2. Полученный код функции fibBU.

```
// Время выполнения = O(n*n). Используются только три значения, а не целый массив - прошлый, текущий, следующий элементы.
static Integer fibBUImproved(int length) {  1 usage
    if (length <= 1) return length;
    int previous = 0;
    int current = 1;
    int next;
    for (int i = 1; i < length; i++) {
        next = previous + current;
        previous = current;
        current = next;
    }
    return current;
}
```

Рисунок 3. Полученный код функции fibBUImproved.

```
Good day! Needed element is on position 10
FibTD thinks, that answer is 34
FibBU thinks, that answer is 34
Improved FibBU thinks, that answer is 34
```

Рисунок 4. Пример выполнения программы.

2. Выполнены примеры для решения второй задачи – нахождение наибольшей возрастающей последовательности. Приведено 4 функции – две

из них (LISBottomUP и LISBottomUPWithSequence) находят число элементов в наибольшей возрастающей последовательности, а другие две (printingWithPrevious и printingWithoutPrevious) двумя разными способами находят элементы возможных последовательностей полученного размера.

```
// Время выполнения = O(n*n).
static Integer LISBottomUp(ArrayList<Integer> numbers) {  1 usage
    ArrayList<Integer> additional = new ArrayList<>();
    for (int i = 0; i < needed; i++) additional.add(i);
    for (int i = 0; i < needed; i++) {
        additional.set(i, 1);
        for (int j = 0; j < i - 1; j++)
            if (numbers.get(j) < numbers.get(i) && additional.get(j) + 1 > additional.get(i))
                additional.set(i, additional.get(j) + 1);
    }
    int ans = 0;
    for (int i = 0; i < needed; i++) ans = (ans >= additional.get(i)) ? ans : additional.get(i);
    return ans;
}
```

Рисунок 5. Полученный код функции LISBottomUP.

```
// Время выполнения = O(n*n).
static ArrayList<Integer> LISBottomUpWithSequence(ArrayList<Integer> numbers) {  1 usage
    ArrayList<Integer> additional = new ArrayList<>();
    ArrayList<Integer> previous = new ArrayList<>();
    for (int i = 0; i < needed; i++) {
        additional.add(i);
        previous.add(i);
    }
    for (int i = 0; i < needed; i++) {
        additional.set(i, 1);
        previous.set(i, -1);
        for (int j = 0; j < i - 1; j++)
            if (numbers.get(j) < numbers.get(i) && additional.get(j) + 1 > additional.get(i)) {
                additional.set(i, additional.get(j) + 1);
                previous.set(i, j);
            }
    }
    int ans = 0;
    for (int i = 0; i < needed; i++) ans = (ans >= additional.get(i)) ? ans : additional.get(i);
    System.out.println("Function with previous counted elements:");
    printingWithPrevious(ans, additional, previous, numbers);
    System.out.println("Function without previous counted elements:");
    printingWithoutPrevious(ans, additional, numbers);
    return additional;
}
```

Рисунок 6. Полученный код функции LISBottomUPWithSequence.

```
static void printingWithPrevious(int ans, ArrayList<Integer> additional, ArrayList<Integer> previous, ArrayList<Integer>
    ArrayList<Integer> reincarnate = new ArrayList<>();
    for (int i = 0; i < ans; i++) reincarnate.add(i);
    int k = 0;
    for (int i = 1; i < needed; i++)
        if (additional.get(i) > additional.get(k)) k = i;
    int j = ans - 1;
    while (k > 0) {
        reincarnate.set(j, k);
        j--;
        k = previous.get(k);
    }
    for (int i = 0; i < ans; i++) System.out.print(numbers.get(reincarnate.get(i)) + ", ");
    System.out.println();
}
```

Рисунок 7. Полученный код первой находящей функции.

```
static void printingWithoutPrevious(int ans, ArrayList<Integer> additional, ArrayList<Integer> numbers) {  1 usage
    int element = ans;
    ArrayList<Integer> values = new ArrayList<>();
    for (int i = needed - 1; i > -1; i--)
        if (additional.get(i) == element && (values.isEmpty() || values.get(values.size() - 1) > numbers.get(i))) {
            values.add(numbers.get(i));
            element--;
        }
    for (int i = ans - 1; i > -1; i--) System.out.print(values.get(i) + ", ");
    System.out.println();
}
```

Рисунок 8. Полученный код второй находящей функции.

```
Good day! elements are:[111, 220, 203, 168, 11, 60, 44, 128, 217, 131, 129, 75, 202, 186, 103, 101, 106, 187, 74, 212]
LISBottomUp thinks that biggest sequence is 6
Function with previous counted elements:
11, 44, 75, 103, 106, 212,
Function without previous counted elements:
11, 44, 75, 101, 187, 212,
LISBottomUpWithSequence gave indexes to elements: [1, 1, 2, 2, 1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 3, 6]
```

Рисунок 9. Пример выполнения второго раздела примеров.

3. Выполнены примеры для решения третьей задачи – нахождение наибольшей возрастающей последовательности. Приведено три функции: reincarnate (Находит, какие элементы были использованы в подсчётах метода одинарных предметов), knapSackWithoutRepetitions (Метод, считающий сумму цен за предметы, которые можно положить только в одном экземпляре), knapSackWithRepetitions (Метод, считающий сумму цен за предметы, которые можно положить в неограниченном количестве).

```
// Нахождение решения без использования повторений. Затрачиваемое время = O(n * allowedWeight) (без учёта reincarnate)
static void knapsackWithoutRepetitionsBU(int[] weights, int[] values) {  1 usage
    int[][] dpTable = new int[allowedWeight + 1][weights.length + 1];
    for (int i = 0; i <= weights.length; i++) dpTable[0][i] = 0;
    for (int weight = 0; weight <= allowedWeight; weight++) dpTable[weight][0] = 0;
    for (int item = 1; item <= weights.length; item++) {
        for (int weight = 1; weight <= allowedWeight; weight++) {
            dpTable[weight][item] = dpTable[weight][item - 1];
            int currentWeight = weights[item - 1];
            if (currentWeight <= weight)
                dpTable[weight][item] = Math.max(dpTable[weight][item], dpTable[weight
                        - currentWeight][item - 1] + values[item - 1]);
        }
    }
    List<Integer> reincarnation = reincarnate(dpTable, weights, values);
    System.out.println("Method without repetitions counted price = " + dpTable[allowedWeight][weights.length]);
    System.out.println("Included elements are:");
    for (int i = 0; i < reincarnation.size(); i++)
        if (reincarnation.get(i) == 1)
            System.out.println("Element" + (i+1) + ", which weights " + weights[i] + " and costs " + values[i]);
}
```

Рисунок 10. Полученный код способа без повторения товаров.

```java
// Нахождение решения с использованием повторений. Затрачиваемое время = O(n * allowedWeight)
static void knapsackWithRepetitionsBU(int[] weights, int[] values) {  1 usage
    int[] dpTable = new int[allowedWeight + 1];
    for (int i = 0; i < weights.length; i++)
        for (int weight = weights[i]; weight <= allowedWeight; weight++)
            dpTable[weight] = Math.max(dpTable[weight], dpTable[weight - weights[i]] + values[i]);
    System.out.println("Method with repetitions counted price = " + dpTable[allowedWeight] + "\n");
}
```

Рисунок 11. Полученный код способа с повторением вещей.

```java
// Восстановление решения с учетом повторяющихся предметов.
public static List<Integer> reincarnate(int[][] dpTable, int[] weights, int[] values) {  1 usage
    List<Integer> solution = new ArrayList<>();
    int remainingWeight = allowedWeight;
    int currentItem = weights.length;
    // Обратный проход для восстановления решения
    for (int i = weights.length - 1; i >= 0 && remainingWeight > 0 && currentItem > 0; i--) {
        int currentWeight = weights[i];
        if (dpTable[remainingWeight][currentItem] == dpTable[remainingWeight - currentWeight][currentItem - 1] + values[i]) {
            solution.add(1);  // Этот предмет включён в рюкзак.
            remainingWeight -= currentWeight;
        } else solution.add(0);  // Этот предмет не включён в рюкзак.
        currentItem--;
    }
    Collections.reverse(solution);
    return solution;
}
```

Рисунок 12. Функция, подсчитывающая использованные предметы.

```
Good day! It's KnapSack program!
Sack can keep weight = 10
1 thing weights 6 and costs 30
2 thing weights 3 and costs 14
3 thing weights 4 and costs 16
4 thing weights 2 and costs 9
Method without repetitions counted price = 46
Included elements are:
Element1, which weights 6 and costs 30
Element3, which weights 4 and costs 16
Method with repetitions counted price = 48
```

Рисунок 13. Полученный результат выполнения программы.

Вывод: в ходе выполнения практической работы были рассмотрены примеры динамического программирования на трёх типах задач: нахождение значения числа Фибоначчи на требуемой позиции, нахождение наибольшей возрастающей последовательности, нахождение цены за заполненный вещами рюкзак. В процессе выполнения работы проведено ознакомление с динамическим программированием, чья суть заключается в решении сложных задач путём разбиения их на более простые подзадачи.